НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування і спеціалізованих комп‘ютерних систем

«До захисту допущено»

Завідувач кафедри

_____ Віталій РОМАНКЕВИЧ
(підпис) (ініціали, прізвище)

“___” червня 2020 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Системне програмування»

зі спеціальності **123 «Комп'ютерна інженерія»**

на тему: «Паралельно розподілений синтаксичний аналізатор»

Виконав: студент IV курсу, групи КВ-62
(шифр групи)

Вовчок Олексій Володимирович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник к.т.н., доц. каф. СПСКС, Зорін Ю. М. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант з нормоконтролю, доц. каф. СПСКС, к.т.н. Клятченко Я. М. _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали) __ (підпис)

Рецензент _____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2020 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування і спеціалізованих комп‘ютерних систем

Рівень вищої освіти – перший (бакалаврський)

Спеціальність 123 «Комп’ютерна інженерія»

Освітньо-професійна програма «Системне програмування»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Віталій РОМАНКЕВИЧ
(підпис) (ініціали, прізвище)

«___» червня 2020 р.

ЗАВДАННЯ
на дипломний проєкт студента
Вовчка Олексія Володимировича
(прізвище, ім’я, по батькові)

1. Тема проєкту «Паралельно розподілений синтаксичний аналізатор»,
керівник проєкту Зорін Юрій Михайлович, к.т.н., доц. каф. СПСКС,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «29» квітня 2020 р. №_____

2. Термін подання студентом проєкту _____

3. Вихідні дані до проєкту див. Технічне завдання.

4. Зміст пояснювальної записки

Додаток 2. Структура проєкту паралельного синтаксичного аналізатора. Схема структурна……..……..……55

Додаток 3. UML-діаграма взаємозв'язків класів. Діаграми класів……56

Додаток 4. Алгоритм автоматизованого тестування синтаксичного аналізатора. Схема алгоритму ……..……57

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслеників, плакатів, презентацій тощо)

- алгоритм паралельного синтаксичного аналізатора (схема алгоритму);

- структура проєкту паралельного синтаксичного аналізатора (схема структурна);

- UML-діаграма взаємозв'язків класів (діаграма класів);

- алгоритм автоматизованого тестування паралельного синтаксичного аналізатора (схема алгоритму).

6. Консультанти розділів проєкту*

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Клятченко Я.М., доцент		

7. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1.	Вивчення літератури за тематикою проекту	15.11.2019	
2.	Розроблення та узгодження технічного завдання	30.11.2019	
3.	Аналіз існуючих рішень	05.02.2020	
4.	Підготовка матеріалів першого	05.04.2020	

* Консультантом не може бути зазначено керівника дипломного проекту.

	розділу дипломного проекту		
5.	Підготовка матеріалів другого розділу дипломного проекту	03.05.2020	
6.	Підготовка матеріалів третього розділу дипломного проекту	05.05.2020	
7.	Підготовка матеріалів четвертого розділу дипломного проекту	10.05.2020	
8.	Підготовка графічної частини дипломного проекту	12.05.2020	
9.	Оформлення документації дипломного проекту	15.05.2020	

Студент _____ Олексій ВОВЧОК
(підпис) (ініціали, прізвище)

Керівник проєкту _____ Юрій ЗОРІН
(підпис) (ініціали, прізвище)

АНОТАЦІЯ

Кваліфікаційна робота включає пояснювальну записку (53с., 16 рис., 4 додатки).

У роботі було розроблено автоматичний генератор на основі низхідної LL(*)-граматики, що будує відповідний паралельний синтаксичний аналізатор. Таким чином досягається приріст швидкодії за рахунок використання незадіяних ядер процесора.

Даний проект дозволяє:

- автоматично будувати синтаксичний аналізатор із задання контекстно-вільної граматики з допомогою БНФ, поданої s-виразами мови Common Lisp;
- розпаралелювати задачу синтаксичного аналізу на задану кількість потоків виконання;
- переключатися між режимом об’єднання там розділення етапів лексичного та синтаксичного аналізаторів;
- проводити виміри швидкодії та використаної пам’яті для визначення найкращої політики розбору заданої мови;
- генерувати тестові рядки заданої мови із зазначеними обмеженнями;
- перевіряти паралельний синтаксичний аналізатор відносно послідовного на згенерованих рядках.

В ході виконання дипломного проєкту:

- проаналізовано автоматизованої способи побудови синтаксичних аналізаторів;
- проведено аналіз теоретичних напрацювань використання паралелізму для задачі синтаксичного аналізу;
- приведені необхідні схеми та документація, підведені підсумки щодо проведеної роботи.

Ключові слова: синтаксичний аналіз, паралельні обчислення, контекстно-вільна граматика, Common Lisp.

ABSTRACT

Qualification work includes an explanatory note (56p., 40 fig., 4 appendices).

In this project, an automatic generator of a parallel parser was developed from given top-down LL(*)-grammar. An increase in performance is achieved by means of the use of idle CPU cores.

This project allows to:

- automatically build a parser from the given context-free grammar BNF, given in s-expressions of the Common Lisp language;
- parallelize the parsing problem for a given number of execution threads;
- switch between the mode of combining and the separation of stages of lexical and syntactic analyzers;
- measure the speed and memory used to determine the best policy for parsing a given language;
- generate test lines of a given language with the specified restrictions;
- check the parallel parser against sequential one on the generated strings.

During the implementation of the project:

- automated methods of constructing parsers are analyzed;
- analysis of theoretical developments in the use of parallelism targeting the problem of parsing;
- the necessary schemes and documentation are given, the results concerning the carried-out work are summed up.

Keywords: parsing, parallel computing, context-free grammar, Common Lisp.

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	Примітки
	А4	ІАЛЦ.045490.002 ТЗ	Паралельно розподілений	4		
			синтаксичний			
			аналізатор			
			Технічне завдання			
	А4	ІАЛЦ.045490.003 ТП	Паралельно розподілений	2		
			синтаксичний			
			аналізатор			
			Відомість технічного			
			проекту			
	А4	ІАЛЦ.045490.004 ПЗ	Паралельно розподілений	53		
			синтаксичний			
			аналізатор.			
			Пояснювальна записка			
	А4	ІАЛЦ.045490.005 Д1	Алгоритм паралельного	1		
			синтаксичного			
			аналізатора рядка.			
			Схема алгоритму			

Змін.	Арк.	№ докум.	Підпис	Дата	ІАЛЦ.045490.001 ОА		
Розробив		Вовчок О. В.			Паралельно розподілений синтаксичний аналізатор **Опис альбому**	Літ.	Аркуш 1 / Аркушів 2
Перевірив		Зорін Ю. М.					
Консульт.						КПІ ім. Ігоря Сікорського, ФПМ КВ-62	
Н. контроль		Клятченко Я.М.					
Зав. каф.		Романкевич В.О.					

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	Примітки
	А4	ІАЛЦ.045490.006 Д2	Структура проєкту	1		
			паралельного			
			синтаксичного			
			аналізатора.			
			Схема структурна			
	А4	ІАЛЦ.045490.007 Д3	UML-діаграма	1		
			взаємозв’язків класів.			
			Діаграма класів			
	А4	ІАЛЦ.045490.008 Д4	Алгоритм	1		
			автоматизованого			
			тестування паралельного			
			синтаксичного			
			аналізатора.			
			Схема алгоритму			
		Диск CD-ROM	Текст пояснювальної	1		
			записки.			
			Графічний матеріал			

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ГАЛУЗЬ РОЗРОБКИ

Назва розробки: «Паралельно розподілений синтаксичний аналізатор».

Галузь застосування: інформаційні технології.

2 ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання роботи першого (бакалаврського) рівня вищої освіти, затверджене кафедрою системного програмування і спеціалізованих комп'ютерних систем Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського».

3 МЕТА І ПРИЗНАЧЕННЯ РОБОТИ

Метою даного проєкту є створення автоматичного генератора паралельного синтаксичного аналізатора за заданою LL(*)-граматикою та порівняння його швидкодії з відповідним послідовним синтаксичним аналізатором.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелом інформації є технічна та науково-технічна література, технічна документація, публікації в періодичних виданнях та електронні статті у мережі Інтернет.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до програмного продукту, що розробляється

- сумісність з операційною системою Linux;
- можливість сумісного застосування із сучасними автоматизованими системами побудови трансляторів;
- обробка LL(*)-граматик;
- можливість автоматизованої побудови коду синтаксичного аналізатора
- можливість паралельного виконання із вказуванням кількості потоків-аналізаторів.

5.2. Вимоги до апаратного забезпечення

- оперативна пам’ять: 1 Гб;
- процесор MIMD архітектури.

5.3 Вимоги до програмного забезпечення користувача

- операційна система сімейства Debian;
- Steel Bank Common Lisp компілятор.

6 ЕТАПИ РОЗРОБКИ

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1.	Вивчення літератури за тематикою проекту	15.11.2019	
2.	Розроблення та узгодження технічного завдання	30.11.2019	
3.	Аналіз існуючих рішень	05.02.2020	
4.	Підготовка матеріалів першого розділу дипломного проекту	05.04.2020	
5.	Підготовка матеріалів другого розділу дипломного проекту	03.05.2020	
6.	Підготовка матеріалів третього розділу дипломного проекту	05.05.2020	
7.	Підготовка матеріалів четвертого розділу дипломного проекту	10.05.2020	
8.	Підготовка графічної частини дипломного проекту	12.05.2020	
9.	Оформлення документації дипломного проекту	25.05.2020	

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	Примітки
	А4	ІАЛЦ.045490.004 ПЗ	Паралельно	53		
			синтаксичний			
			аналізатор.			
			Пояснювальна записка			
	А4	ІАЛЦ.045490.005 Д1	Алгоритм паралельного	1		
			синтаксичного			
			аналізатора рядка.			
			Схема алгоритму			
	А4	ІАЛЦ.045490.006 Д2	Структура проєкту	1		
			паралельного			
			Синтаксичного			
			аналізатора.			
			Схема структурна			
	А4	ІАЛЦ.045490.007 Д3	UML-діаграма	1		
			взаємозв’язків класів.			
			Діаграми класів			

Змін.	Арк.	№ докум.	Підпис	Дата	ІАЛЦ.045490.003 ТП		
Розробив		Вовчок О. В.			Паралельно розподілений синтаксичний аналізатор	Літ.	Аркуш 1 / Аркушів 2
Перевірив		Зорін Ю. М.					
Консульт.					**Відомість технічного проєкту**	КПІ ім. Ігоря Сікорського, ФПМ КВ-62	
Н. контроль		Клятченко Я.М.					
Зав. каф.		Романкевич В.О.					

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	Примітки
	А4	ІАЛЦ.045490.008 Д4	Алгоритм	1		
			автоматизованого			
			тестування			
			синтаксичного			
			аналізатора.			
			Схема алгоритму			
		Диск CD-ROM	Текст пояснювальної	1		
			записки.			
			Графічний матеріал			

					ІАЛЦ.045490.003 ТП	Арк.
Змін.	Арк.	№ докум.	Підпис	Дата		2

Пояснювальна записка
до дипломного проєкту

на тему: «Паралельно розподілений синтаксичний аналізатор»

Київ – 2020 року

ЗМІСТ

ДОДАТКИ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

СА – синтаксичний аналізатор.

САГСА – система автоматичної генерації СА.

БНФ– нотація Бекуса-Наура (англ. Backus-Naur form, BNF).

РБНФ – розширена БНФ (англ. Extended BNF, EBNF).

АМП – автомат з магазинною пам’яттю (англ. Pushdown automaton).

КЗ – контекстно-залежна (граматика, мова).

КВ – контекстно-вільна (граматика, мова).

ГВР– граматика виразів розбору, (англ. Parsing expression grammar, PEG).

ОС – операційна система.

ІСМ – ієрархічна специфікація мови.

ГЗМП – граф залежності маркерних пар.

ВСТУП

Розвиток мікропроцесорів досяг рівня побудови кількох процесорів на одному кристалі, що забезпечує паралельне виконання мікрокоманд. З іншого боку, розвиток ІТ галузі на сьогодні призводить до створення обширних за обсягом програмних рішень. Тим часом командна розробки таких проєктів призводить до частих компіляцій. Тоді якщо компілятор мови програмування не підтримує розпаралелювання, то буде використана лише певна частина обчислювальної потужності процесора, хоча можливе прискорення процесу компіляції.

Одна з найбільш затратних операцій компіляції або інтерпретації програм є побудова дерев розбору синтаксичним аналізатором. Такі аналізатори у своїй більшості представлені системами, що забезпечують можливість побудувати синтаксичний аналізатор відповідно до вхідної граматики. Їх називають системами побудови трансляторів (англ.: “compiler compiler”). Таким програмним системам передають на вхід описану в деякій визначеній формі, скажімо, БНФ, граматику мови, і результатом їх роботи є побудований СА деякою (зазвичай з високою швидкодією) мовою програмування (C, Java та ін.). Тому розроблювана САГСА повинна бути сумісною із вже існуючими рішеннями.

Ще одним застосуванням для розпаралелювання СА є розбір популярних на сьогодні текстових форматів представлення даних (XML, JSON, YAML та інші). Такі мови мають простіші граматики в порівнянні з сучасними мовами програмування і такі файли зазвичай мають більший розмір, оскільки в сучасному програмуванні заохочується принцип «розділяй і володарюй», а файли даних можуть сягати значних розмірів, що створює передумови для орієнтації паралельного СА саме для таких граматик.

Також зважаючи на те, що розвиток процесорів відбувається не скільки у бік рівня мініатюризації, скільки в бік зростання кількості ядер, а, єдиним

способом ефективно справлятись із постійно зростаючою кількістю даних для обробки є застосування паралелізму скрізь, де це можливо.

Зрозуміло, що повністю паралельне виконання синтаксичного аналізу складно піддається реалізації, враховуючи природню послідовність цієї задачі та орієнтацію поточних рішень на послідовне виконання. Тому можна припустити, що не всі мови будуть однаково ефективно підлягати паралельному синтаксичному аналізу, тому необхідно, крім реалізації власне САГСА, мати систему перевірки швидкодії для заданої мови із визначенням, хоча б емпіричним шляхом, ефективності отриманого рішення для різних розмірів задачі.

1 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ОБҐРУНТУВАННЯ ТЕМИ ДИПЛОМНОГО ПРОЄКТУ

1.1 Теорія формальних граматик

Для подальшого обґрунтування розглянемо загальновідомі поняття та класифікації з теорії формальних граматик.

Граматика – спосіб опису формальної мови, що виявляється у виділенні деякої підмножини із множини всіх можливих комбінацій символів деякого скінченного алфавіту. Розрізняються породжувальні граматики, що задають правила, за якими можна побудувати будь-яке речення мови, та аналітичні, що задають правила, за якими можна визначити, чи входить конкретне речення в задану до алфавіту заданої мови. Поняття формальних граматик були введені американським вченим, математиком та філософом, Н. Хомським у 50-тих роках XX сторіччя в його класифікації мов.

Формальна граматика – це четвірка G = { N , T , P , S }, де:

- T – множина термінальних символів, терміналів. Термінали задаються довільною непустою скінченною множиною символів, що задають алфавіт мови.
- N – множина нетермінальних символів, нетерміналів.
- S – аксіома – спеціально виділений нетермінал, з якого починається опис граматики. Зазвичай вказують першим.
- P — правила виводу, скінченна підмножина множини.

Хомський класифікував формальні мови у своїй книзі «Синтаксичні структури», увівши при цьому чотири класи формальних граматик.

Граматики типу 0

Мови, що породжуються граматиками типу 0, називають необмеженими. До них належать всі природні мови. Кожна граматика типу 0 породжує мову, яку може розпізнати машина Тюринга, та навпаки: для кожної мови, яку може розпізнати машина Тюринга, існує граматика типу 0, яка здатна її породити [1].

Граматики типу 1

До даного типу належить підмножина граматик типу 0, в яких правила виводу обмежено правилами вигляду

$$\alpha X \beta \rightarrow \alpha \gamma \beta , \qquad (1)$$

де X – нетермінал, а α, γ, β – рядки, що можуть складатись із термінальних (T), нетермінальних (N) символів. Контекстно-залежні (КЗ) граматики породжують контекстно-залежні мови; тобто, кожна КЗ граматика породжує КЗ мову, і навпаки – для кожної КЗ мови існує КЗ граматика, що її породжує. Контекстно-залежні мови розпізнаються недетермінованою лінійно-обмеженою машиною Тюринга; тобто, недетермінованою машиною Тюринга, довжина стрічки якої обмежена [1].

Граматики типу 2

В кожному правилі граматики типу 2 (контекстно-вільної граматики) з лівої сторони знаходиться один нетермінальний символ, а з правої сторони — можливо порожня послідовність термінальних та нетермінальних символів. Тобто, на правила виводу накладено обмеження Граматики типу 2 мають лише правила виду

$$X \rightarrow \gamma , \qquad (2)$$

де X – нетермінальний символ, а γ – рядок, що складається з терміналів та нетерміналів. Контекстно-вільні (КВ) граматики породжують КВ мови, і для кожної КВ мови існує КВ граматика, що її породжує. Контекстно-вільні мови можуть бути розпізнані недетермінованими автоматами з магазинною пам'яттю. Контекстно-вільні мови широко використовуються в побудові синтаксичних властивостей мов програмування [1].

Граматики типу 3

Граматики третього типу називають регулярними. До них належать граматики другого типу, правила виводу яких обмежено лише правилами вигляду:

$$X \to \gamma Y \tag{3}$$

або

$$X \to Y\gamma, \tag{4}$$

де γ – термінальний символ, X – нетермінал.

Граматику, обмежену лише правилами відповідно до формули 3, називають праволінійною, відповідно до формули 4 – ліволінійною. Для кожної праволінійної граматики існує ліволінійна граматика, та навпаки. Регулярні граматики породжують регулярні мови, і для кожної регулярної мови існує регулярна граматика, що її породжує. Регулярні мови також описують класичними регулярними виразами. Регулярні мови можна розпізнати скінченними автоматами [1]. Такі мови широко застосовуються для пошуку і заміни фрагментів тексту, а також для задання структури лексичного аналізатора мови програмування.

1.2 Класифікація СА

Класичний синтаксичний аналізатор мови програмування створюється для побудови синтаксичного дерева розбору із вихідної, у переважній більшості випадків, контекстно-вільної мови, що може бути реалізовано на базі автомата з магазинною пам’яттю. У побудові СА існує ряд нюансів, що відображається в наступній класифікації:

1. За напрямком обходу послідовності:
   1.1. зліва направо;
   1.2. справа наліво;
   1.3. довільно.
2. За стратегією СА
   2.1. СА з висхідною стратегією;
   2.2. СА з низхідною стратегією;
   2.3. СА зі змішаною стратегією.
3. За кількістю переглянутих терміналів для прийняття рішення
   3.1. 1 символ;
   3.2. k символів.
4. За використанням повернень
   4.1. без повернень (передбачаючий);
   4.2. з поверненнями.
5. За відновленням після знайденої помилки
   5.1. з побудовою результуючого дерева розбору із помилковим піддеревом;
   5.2. з аварійним завершенням СА після знайдення помилки.

Перша класифікація вказує на напрямок зчитування аналізатором вихідної послідовності. Оскільки так історично склалось, що мови програмування майже повністю всі читаються людьми зліва направо, то відповідний підхід набув найбільш широкого поширення. Частково можна вважати стратегію паралельного зчитування фрагментів вихідної послідовності як довільну, відповідно до класифікації, але все ж фрагменти все одно можуть зчитуватись в певному порядку.

Друга класифікація розділяє СА за підходами до послідовності побудови дерева розбору. Підхід цей визначає послідовність розгортки нетерміналів, з яких складається нетермінал, що аналізується в даний момент. Низхідна

стратегія (англ. “top-down”) реалізовує побудову дерева розбору з найвищої форми до найнижчої, а висхідна (англ. “bottom-up”) – навпаки. Для певних мов різниця цих підходів може впливати на швидкодію.

Третя класифікація розділяє перевірку способів розгортки нетерміналів за перевіркою 1 символу, або k символів, де k > 1.

Четверта класифікація розділяє СА за політикою обробки альтернатив. Передбачаючий СА – аналізатор, що при необхідності вибору з кількох заданих альтернатив правила може однозначно вирішити, по якій альтернативі має розкриватися поточне правило. Цей вибір (передбачення) відбувається на основі перевірки k вхідних символів. Його алгоритмічна складність в такому випадку лінійна. СА з поверненням – аналізатор, що при необхідності вибору з кількох заданих альтернатив намагається розкрити кожну альтернативу послідовно в порядку їх задання. Тобто, разі виникнення помилки повертається для перевірки наступної альтернативи і. Тому алгоритмічна складність найгіршого випадку експоненційна.

За п’ятою класифікацією розділяють поведінку СА при виникненні ситуації знайденої невідповідності переданого рядка заданій мові. Оскільки результатом роботи СА є дерево розбору, то теоретично можна спробувати виправити помилку та побудувати дерево розбору. Для цього виконують аналіз заданої мови або доповнюють її властивостями з відновлення дерева розбору. Але зважаючи на свою простоту реалізації набагато ширше використовують підхід видачі помилки без продовження побудови дерева розбору.

1.3 Автоматизовані СА

На сьогодні існують наступні методи автоматичної побудови СА:

- генератори парсерів
- парсер-комбінатори

Генератор парсера (англ. “parser generator”) – програмний комплекс, що призначений для побудови програмного коду синтаксичного і/або лексичного аналізаторів з певного виду формального опису граматики, зазвичай — БНФ або РБНФ.

Парсер-комбінатор – функція вищого порядку, що бере як кілька інших парсер-комбінаторів та як результат будує новий СА, що є логічним поєднанням вхідних. В переважній більшості випадків, парсер-комбінатори дозволяють описувати СА за допомогою мови програмування, без необхідності використовувати додаткові види нотації такі, як БНФ [3]. Оскільки парсер-комбінатор використовується мовою програмування, то результуючий СА можна доповнити контекстно-залежними властивостями мови, наприклад обробку табуляцій в мові Python. Парсер-комбінатор є більш простим, але менш швидкодіючим інструментом в порівнянні з генератором парсерів. Недоліки парсер-комбінаторів:

- менша швидкодія;
- менша варіативність та гнучкість (підтримують лише низхідний аналіз LL(1) граматик);
- процес розробки не піддається автоматизації (хоча рівень абстракції може це компенсувати).

Оскільки генератори парсерів позбавлені таких недоліків, для реалізації проєкту обрано алгоритм їх роботи.

1.4 Обґрунтування теми дипломного проєкту

Як вже було сказано раніше, має сенс завантажити всю обчислювальну потужність процесора для задачі синтаксичного аналізу. Цього можна досягти в кілька способів:

- паралельно аналізувати кілька наборів вхідних даних (файлів);
- аналізувати вхідну послідовність автоматом СА порівну розділені частини вхідної послідовності.

На даний час існує широкий набір САГСА для практично будь-якого типу СА, але не кожен з них надає можливість для паралельного аналізу кількох файлів. Такі САГСА можуть не підтримувати запуск кількох процесів СА на одній і тій же граматиці, оскільки може бути заблокований загальний ресурс таких СА – таблиця автомата СА. Однак така проблема вже не така поширена, як це можна було спостерігати раніше, і тому такі спільні ресурси або копіюють для кожного запущеного СА або використовують механізми синхронізації.

Що ж до іншої можливості паралельного СА – аналізу кількох частин послідовності і паралельного розбору, то такі СА не набули широкого вжитку. Теоретично обґрунтована база все ж існує, зокрема для лексичного аналізатора [6], СА з низхідною стратегією [7], а також з висхідною LR(*) [8].

Даний проєкт застосовує названі теоретичні відомості для побудови і порівняння двох названих стратегій паралелізації низхідного LL(*), а також порівняння цього СА з послідовним аналогом.

1.5 Огляд існуючих аналогів

Оскільки таких САГСА, що б підтримували побудову специфічно паралелізованих СА з відкритим програмним кодом не знайдено, то проведемо порівняння хоча б з існуючими САГСА з генерацією послідовних СА.

На даний час існує достатня кількість САГСА для граматики кожної мови програмування практично для кожного типу СА. Наприклад, до найбільш відомих існуючих САГСА належать

- генератори ЛА (flex, lex);

- генератори парсерів (yacc, bison);
- генератори як ЛА, так і СА (ANTLR, PEG.js).

Програмні комплекси ANTLR та PEG.js здатні генерувати як лексичну, так і синтаксичну частину аналізаторів. Для представлення вхідної граматики ANTLR використовує РБНФ, натомість PEG.js – формалізм граматки виразів розбору (ВР-граматика, англ. Parsing expression grammar) [4]. Flex та lex використовуються в комплекті з yacc або bison і таким чино утворюють висхідний LR(*) СА на мові С. Вони використовують спеціальний синтаксис, що реалізує семантику одночасного задання БНФ та вставленого коду на мові С, що описує обробку нетермінала [5].

Парсер-комбінатори представлені бібліотеками Parsec, що написана на мові Haskell та SMUG – на мові Common Lisp.

1.6 Відмінності від схожих за назвою проєктів

Паралельний GCC – дослідницький проєкт, що націлений на забезпечення паралелізму в сучасних компіляторах мов, що підтримує GCC. Як зазначено на головній сторінці проєкту, власне паралелізація застосована для оптимізації не парсингу або збірки, а натомість виконується розпаралелювання оптимізатора коду, що генерується компілятором [10].

Подібні проєкти називають паралелізуючими компіляторами (англ. “parallelizing compiler”). Це програма, що знаходить паралелізм у послідовній програмі генерує відповідний код для комп’ютера з підтримкою паралельного виконання програми, що компілюється. Такі компілятори шукають явно паралельні мовні конструкції, такі як копіювання масивів або цикли, виконання яких можна паралелізувати.

2 ТЕОРЕТИЧНЕ ОБҐРУНТУВАННЯ ПАРАЛЕЛЬНОГО СИНТАКСИЧНОГО АНАЛІЗУ

2.1 Паралелізація лексичного аналізу

2.1.1 Загальна характеристика

Лексичний аналіз проводиться перед власне СА та трансляцією, і існує загальноприйнята думка, що це досить легка та менш затратна робота в порівнянні з наступними фазами. Хоча це справедливо для більшості мов програмування, але таки і для них ЛА та СА є порівнюваними відносно затрачених ресурсів, тому важливо приділити ЛА достатньо уваги для помітного виграшу від використання паралелізму. Крім того, для реалізації ЛА необхідно застосувати певні евристичні передбачення щодо властивостей мови, які не передані явно граматикою мови, але при цьому несуть певні неочевидні оптимізаційні доповнення, важливі для паралельного вирішення задачі ЛА. Однак для досягнення цієї мети необхідно подолати кілька нетривіальних технічних труднощів.

У цьому розділі представлено досить загальну схему паралелізації лексичного аналізу, яка може бути застосована до більшості мов програмування. Надамо визначення, необхідні для опису паралелізації ЛА.

Лексика мови описується лексичною граматикою, яка передбачає символи, присутні у вхідному потоці, як термінальний алфавіт. Зазвичай лексичний синтаксис можна проаналізувати за допомогою скінченного автомата (англ. Finite state machine, FSM) [2], на відміну від автомата з магазинною пам’яттю, який застосовується для СА КВ-мов.

Введемо деяку базову термінологію , яка адаптує деякі загальні терміни до сфери цього розділу.

Лексема – це послідовність символів, що відповідає дійсному реченню граматики лексики мови (наприклад, вбудований ідентифікатор, зарезервоване

ключове слово, оператор). Її форма залежить від лексичної частини визначення мови, і вона може бути розпізнана скінченним автоматом.

Рядок – це лексема, побудована у вигляді послідовності символів, укладених у пару роздільників, як правило, одинарних або подвійних лапок. Він не може містити жодних інших роздільників такого ж типу без належного виходу (наприклад, префіксація їх символом \). Рядки можуть містити контрольні символи (наприклад, нові рядки).

Коментар – це послідовність символів, розділених спеціальними символами відповідно до правил, залежних від мови, і не відповідає лексемі. У процесі ЛА коментар повинен розпізнаним та відкинутим (або якщо необхідно, то збереженим, але в такій ситуації необхідна модифікація наступних етапів розбору з урахуванням такого спеціального випадку). Багато мов використовують різні маркери для однорядкових та багаторядкових коментарів.

Мета ЛА – розпізнати лексеми у заданому символьному потоці та генерувати послідовність лексем, видаляючи коментарі. Лексична граматика, зважаючи на те, що вона, як правило, визначає спрощену мову, є більш природним кандидатом для ефективної паралелізації, ніж синтаксичний аналіз, про що буде сказано у відповідному розділі.

Однак для досягнення цієї мети необхідно вирішити два питання. Перш за все, розбиття вихідного тексту випадковим чином на чистини, які обробляються паралельними обробниками, може розділити лексему на різні сегменти. Таким чином, результати, отримані ЛА, що працюють над сусідніми фрагментами, повинні бути узгоджені для вирішення цього питання. По-друге, стосується появи дуже довгих розділів коментарів, можливо, довших, ніж чистина рядка, призначена одному обробникові. Серед деяких програмістів звичайно коментувати частини застарілого або тимчасового коду, фактично заважаючи ЛА знати, чи аналізує він частину коментаря чи текст важливу частину тексту програми, у разі відсутності роздільників коментарів у його частині рядка. Це

ще більше посилюється тим, що деякі мови приймають деякі екзотичні синтаксичні правила для коментарів та/або розділових рядків загалом. Наприклад, у скриптових мовах сімейства Perl, PHP, bash можна виділити рядкові константи за допомогою безмежної кількості символів відкриття та закриття за допомогою конструкції Here-document [9].

```
$ cat <<eof1; cat <<eof2
> Hello,
> eof1
> World!
> eof2
```

Рисунок 1 – Використання here-document в bash-інтерпретаторі

Такі конструкції порушують властивості КВ-мов і можуть бути розпізнаними лише послідовними інтерпретаторами відповідних мов, тому це унеможливлює ефективне розпаралелювання СА.

2.1.2 Етап розділення вихідної послідовності

Спочатку вхідна послідовність розділяється на сегменти однакового розміру; швидше за все, однак, такий поділ може порушити цілісність лексеми. Такою лексемою, як правило, буває ідентифікатор, який зазвичай не дуже довгий. Таким чином, часто достатньо розглянути принцип огляду вперед або назад кількох символів, щоб знайти роздільник лексеми (тобто пробіл або інший роздільник). У таких випадках точка розриву може бути зручно встановлена відразу після кінця лексеми, а межі сегментів оновляться, намагаючись уникнути розділення будь-якої лексеми.

Взагалі, не може бути заранні відомо як далеко знаходиться кінець лексеми від заданої точки; таким чином, тривалість пошуку повинна бути визначена на основі деяких, можливо, залежних від мови евристичних критеріїв, які ні в якому разі не повинні перевищувати в своїй складності перевірку значення кількох символів.

Зазвичай у мовах програмування, початок лексеми можна знайти за роздільником (пробільний символ або символ інфіксної операції). Якщо припустити, що такий роздільник знайдений, можлива неоднозначність початкового стану ЛА зводиться або на початок лексеми, позицію в рядку або позицію в коментарі. Якщо пошук роздільника в межах локального аналізу на початку фрагмента не є достатнім, початковий стан також може відповідати внутрішній точці лексеми рядка.

Після присвоєння даному фрагменту кожен аналізатор проводить обчислення для кожної можливої альтернативи. Але типовий лексичний аналіз із утворенням єдиного списку лексем все ж може відбутися:

- якщо аналізатор зустрічає символ, заборонений в деяких станах автомата ЛА, таким чином організовуючи переривання помилкових обчислень;
- якщо послідовність символів викликає розпізнавання лексеми на більш ніж одній альтернативі, при виникненні помилки в усіх, крім однієї лексичний аналіз залишить лише одну альтернативу.

Таким чином, кожен аналізатор послідовності, окрім першого, починається з початкової неоднозначності того, що він знаходиться або:

- на початку належної лексеми;
- багаторядковому коментарі;
- в рядковому літералі.

Аналізатор здійснює обробку двох варіантів: одна з них одночасно обробляє дві останні можливості зі списку вище, а інша – першу. Аналізатори, які починаються так, ніби вони знаходяться на початку лексеми, або в

коментарях з декількох рядків, повинні проаналізувати свій фрагмент одночасно на відповідність лексемам, так і відстежувати позиції відкриття та закриття коментаря, із запам’ятовуванням їх в список. Щоразу, коли вони знаходять завершальний символ багаторядкового коментаря, вони повертаються до стану, де початок лексем для розбору, відслідковуючи точку закриття коментарів. Список позицій відкриття/закриття багаторядкових коментарів використовується на останній фазі, щоб визначити, які частини списку лексем слід зберігати, а які слід відкинути. Аналізатори повинні мати справу з двома можливими причинами неоднозначності.

1. Закриття багаторядкового коментаря в рядковому літералі. Символ закриття багаторядкового коментаря може виникати в багаторядковому рядковому літералі, в одному й тому ж фрагменті, переданого для аналізатора. У цьому випадку неможливо визначити, чи закінчується коментар, розпочатий у попередньому фрагменті або символ закриття коментаря належить до вмісту поточного рядка. Щоб вирішити це, аналізатор продовжує зчитувати потік символів, поки не буде знайдено символ завершення рядкового літерала. Аналізатор запам'ятовує положення закриття рядка, що дозволяє ідентифікувати кінець рядка під час поєднання списків лексем.

2. Закриття багаторядкового літерала в коментарі. Друга неоднозначність може з'явитися, навпаки, коли символ закінчення багаторядкового літерала, який розпочався в одному фрагменті, закінчується в межах однорядкового коментаря або однорядкового літерала, що відмежований одною парою лапок («”»), аналізатор не може встановити, чи є обмежувач, що закриває, кінець раніше перерваного рядка. Аналізатор повинен почати два незалежні обчислення, збираючи послідуючі лексеми у два відповідні списки.

Дві побудови списків лексем можливо об'єднати в єдину, якщо двозначність можна буде вирішити. Це може статися в наступні ситуації.

Аналізатори можуть скоротити два початкові обчислення до одного, усуваючи неоднозначність тоді, коли в одному з двох запусків виникає помилка і відповідне виконання такого аналізатора припиняється.

2.1.3 Етап поєднування часткових результатів паралельних ЛА

Починаючи з першого фрагменту, в списку лексем, що виробляються кожним аналізатором, перевіряється відповідність символів відкриття і закриття багаторядкових рядків або коментарів. Так ідентифікується фактичний вихідний стан аналізу для кожної частини і перевіряється наявність можливих відкритих роздільників у попередніх сегментах, і остаточний список лексем будується об'єднання частин списків лексем, сформованих уздовж правильних обчислень на фрагментах.

2.2 Паралелізація синтаксичного аналізатора

2.2.1 Постановка задачі

Нехай маємо таку постановку задачі: наявна ЕОМ з кількома процесорами та вхідна послідовність з довільним доступом, що належить до контекстно-вільних мов. Для розпаралелювання алгоритму синтаксичного аналізатора використаємо типовий підхід паралельного програмування [посилання], а саме розподілити задачу на незалежні одна від одної в часі частини і поєднати часткові результати. Тоді очевидним рішенням є розподіл вхідних даних на рівні частини між паралельно працюючими синтаксичними аналізаторами. Тоді отримаємо частини вхідного рядка, що, як правило, не відповідатимуть граматиці вхідної мови. Оскільки такий підхід до розпаралелювання не підтримується ні парсерами з низхідною, ні висхідною стратегіями. Це

очевидно, зважаючи на послідовність виконання перевірок та природної послідовності автомата, що реалізує СА [2]. Отже, необхідно створити видозмінений підхід до вирішення такої задачі.

Тож, для дійсного розпаралелювання нам необхідно створити інструмент, який з певного подання мови L(G) зможе згенерувати паралельний алгоритм, що

- здатний з частини вхідного рядка побудувати частину дерева розбору.
- повинен обов'язково мати більшу швидкодію, бажано з максимальною ефективністю відповідно до закону Амдала [13].

З першого пункту можна зробити висновок, що шуканий інструмент є знову-таки послідовним автоматом з магазинною пам'яттю, подібно до поширених послідовних автоматів синтаксичного аналізу, але при цьому має обробляти вхідну послідовність, що може починатись із будь-якого символу T заданої мови L(G). Тому шуканий інструмент повинен мати відмінний від низхідної і висхідної стратегій початковий стан.

Другий пункт диктує важливість виконання алгоритму з мінімально затраченими ресурсами, по-перше, на поєднання часткових дерев розбору і по-друге, на зменшення кількості варіантів часткових дерев розбору, побудованих окремо взятим обробником, до 1, про що буде наголошено пізніше. Відповідно до закону Амдала:

$$S = \frac{1}{p + \frac{1-p}{n}}, \tag{5}$$

де S – прискорення програми відносно послідовного виконання, p – частина, що має виконуватись послідовно, n – кількість процесорів; можна зробити висновок, що при незмінному n необхідно, щоб p було мінімальним. Для паралельного СА послідовною операцією стає поєднання часткових дерев розбору. Тоді зі зростанням вхідного рядка, p буде зменшуватись, оскільки

передбачається, що асимптотична складність послідовної частини логарифмічна, а паралельної – лінійна.

Таким чином маємо таку постановку задачі: необхідно реалізувати алгоритм, що дозволяє побудувати часткове дерево розбору, починаючи з будь-якого термінального символу T граматики G. Інтуїтивно зрозуміло, що існують мови, при синтаксичному аналізі яких недостатньо перевірити значення 1 термінала у випадковому місці речення мови і мати лише 1 варіант побудови часткового дерева розбору. Отже, необхідно створити механізм, який би в залежності від заданих властивостей мови будував інструмент, що:

- будує таблиці залежностей можливого послідовного розташування терміналів, знаходячи при цьому термінали, що забезпечують єдиний варіант подальшого часткового розбору;
- враховує можливість поєднання результуючих часткових дерев розбору в одне ціле.

Відповідно до попередніх міркувань, маємо рівні частини вихідної послідовності, які передаються паралельно запущеним обробникам (див. додаток 1). Тоді у випадковому місці рядка терміналів кожного обробника, крім першого, виконуємо лінійний пошук термінала, що б забезпечував єдиний варіант подальшого розбору. Для першого обробника ми цього не виконуємо, оскільки в нього наявний початок рядка, а отже є можливість провести класичний низхідний розбір із першого стану автомата CA.

2.2.2 PF-множини

Послідовні CA широко використовується для таких задач, як лексичний аналіз, відповідність рядка шаблону тощо. Метод розбору рядка по відстеження відповідного шляху від стартового стану до кінцевого стану не надає очевидних можливостей з паралелізації. Таким чином, Шелл [6] представив метод

виконання ЛА паралельно. Метод ґрунтується на паралельній реалізації скінченного автомата, вимагаючи при цьому попередню обробку вхідного рядка Нижче застосуємо визначений в [6] критерій можливості синтаксичного аналізу регулярних мов, а саме – PF(k) (precede-follow на k символів) мови. Цей критерій можна буде потім застосувати для паралельного СА.

Зараз, на прикладі розглянемо перевірку заданого критерію залежно від кількості переглянутих символів. Для цього визначимо граматику деякої регулярної мови нижче.

```
<decls> ::= <decl> <decls> | ;
<decl> ::= <const-decl>
         | <var-decl>
         ;
<const-decl> ::= 'const' <id> '=' <number> ';' ;
<var-decl> ::= 'var' <idlist> ':' <type> ';' ;
<idlist> ::= <id>
           | <id> ',' <idlist>
           ;
<type> ::= <id>
         | 'array'
         ;
```

Рисунок 2 – Граматика мови для перевірки PF(*) критерію

Перевіримо на цій граматиці можливість PF(1)-аналізу.

Тоді нехай заданий скінченний автомат M для реалізації аналізу даної мови:

$$M = \{Q, \Sigma, \delta, q_0, F\}, \quad (6)$$

де $Q = \{q_0, q_1, ..., q_n\}$ в – множина станів автомата, Σ – алфавіт, δ – функція переходів $Q \times \Sigma \to Q$, q_0 – початковий стан, F – набір допустимих станів.

Тоді для кожного переходу $\delta(q_i, a) = q_j$ ставлять у відповідність множину $\{(b,c) | \delta(q_k, b) = q_i \wedge \delta(q_j, c) = q_m\}$. Таким чином такий перехід ставиться у відповідність множині всіх можливих пар початкових та кінцевих символів (precede-follow set, PFS):

$$PFS^{(1)}(q_i, a, q_j) = \{(b,c) | \delta(q_k, b) = q_i, \delta(q_j, c) = q_m \forall q_k, q_m \in Q\}. \quad (7)$$

Тоді два переходи $PFS^{(1)}(q_i, a, q_j)$ та $PFS^{(1)}(q_k, a, q_m)$ можливо відрізнити, якщо:

$$PFS^{(1)}(q_i, a, q_j) \cap PFS^{(1)}(q_k, a, q_m) = \varnothing. \quad (8)$$

Тоді автомат M матиме PF(1)-розпізнавальні переходи, якщо всі пари переходів на однакових вхідних даних PF(1)-розпізнавальні. Тоді відповідна мова L(M) вважається PF(1)-аналізовною.

Для більшої наочності зобразимо граф переходів автомата аналізатора заданої граматики на рисунку нижче.

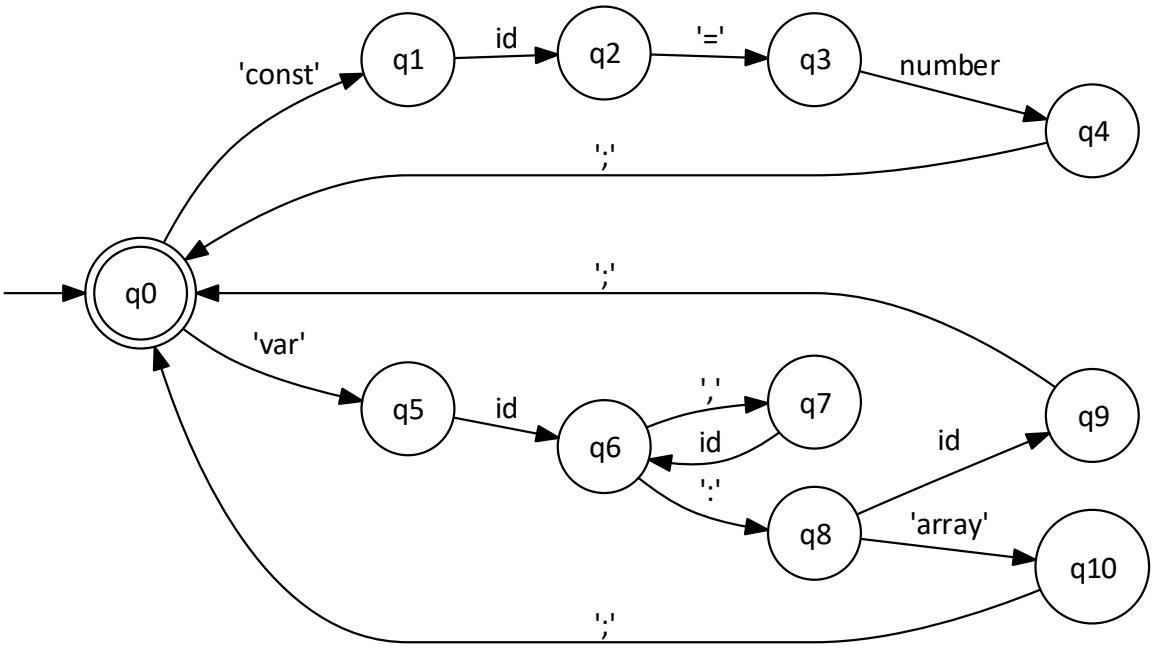


Рисунок 3 – Візуалізація переходів автомата

На основі побудованого графа і цим самим задекларувавши функцію переходів автомата залежно від вхідних даних, побудуємо таблицю множин PF(1) для всіх вхідних даних, що впливають одразу на кілька переходів. Ми можемо ігнорувати вхідні символи, що впливають лише на один перехід, оскільки суть PF-множин полягає у використанні оточуючих символів для вирішення неоднозначності під час розбору послідовності не з початку, а з деякого випадкового місця у заданій послідовності.

Побудувавши вказані множини, можна дійти висновку, що для кожного набору вхідних символів та відповідних йому переходів виконується умова з формули 7, а отже мова, що генерується вказаною граматикою може аналізуватися відповідно до PF(1) критерію. Таким чином, дана мова підлягає розбору не обов’язково від початку, а й від випадкової позиції в рядку, причому для вирішення неоднозначності необхідно переглянути лише один символ.

Таблиця 1 – Множини PF(1)

a	q_i	q_j	$PFS^{(1)}(q_i, a, q_j)$
id	q_1	q_2	{(‘const’, ‘=’)}
id	q_5	q_6	{(‘var’, ‘,’), (‘var’, ‘:’)}
id	q_7	q_6	{(‘,’, ‘:’)}
id	q_8	q_9	{(‘:’, ‘;’)}
‘;’	q_4	q_0	{(number, ‘var’), (number, ‘const’), (number, EOF)}
‘;’	q_9	q_0	{(id, ‘var’), (id, ‘const’), (id, EOF)}
‘;’	q_1	q_0	{(‘array’, ‘var’), (‘array’, ‘const’), (‘array’, EOF)}

2.2.3 Мова Дика

Розглянемо найпростіший приклад, що відображає властивості паралельного СА для контекстно-вільних мов, а не тільки регулярних.

Мова збалансованих дужок, мова Дика (англ. Dyck language) [12], найпростішу з яких можна задати зокрема такою КВ-граматикою, як на рисунку:

```
<S> ::= ('(' <S> ')')+
      | '(' ')'
      ;
```

Рисунок 4 – Граматика мови збалансованих дужок в формі БНФ

Ілюстрація підтримки саме даної мови була вибрана, оскільки вона є найпростішою, що при цьому відображає контекстно-вільні властивості мови, що потребують реалізації за допомогою автомата з магазинною пам’яттю.

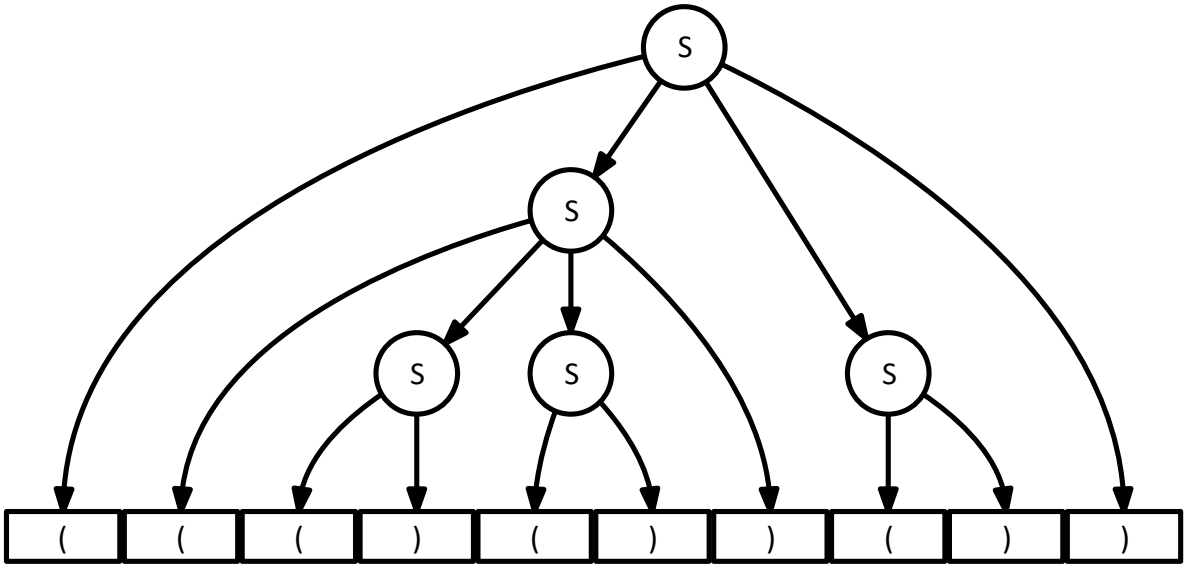


Рисунок 5 – Цілісне дерево розбору

Поділимо задану послідовність із 10 символів на три паралельно і незалежно запущені аналізатори. Таким чином отримаємо три часткові дерева розбору, що містять в собі непоєднані піддерева. На рисунку нижче до термінала S було додано символ «-», щоб показати неповноту вузла дерева розбору. Оскільки в даному випадку достатньо показати, що не вистачає початкової або кінцевої частини, тому достатньо вказати «-» в префіксі, якщо не вистачає «(», і навпаки – суфікс вказує на відсутність «)».

Паралельний СА КВ-граматики можна представити як комбінацію напрацювань для паралельного аналізу мови Дика та PF(*) мови [7].

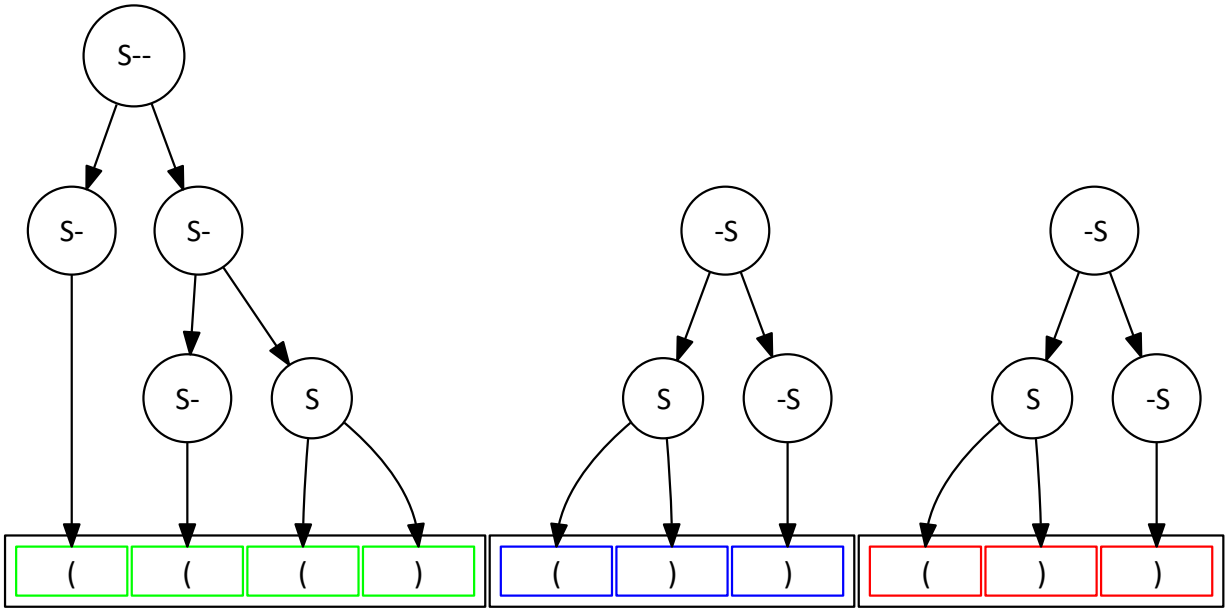


Рисунок 6 – Часткові дерева розбору

Для кожного можливого розриву (з урахуванням змінної пам’яті автомата з магазинною пам’яттю), генерується код для поєднання двох часткових дерев розбору.

Щоб запобігти виникненню нових неоднозначностей, алгоритм поєднання часткових дерев розбору виконується послідовно, починаючи з початкового. Для реалізації поєднання піддерев, можна виділити дві стратегії:

1. Конструктивну. Під час поєднання дерев зв’язки повністю оновлюються, утворюючи нове дерево. Аргументи поєднання залишаються незмінними. Іншими словами, поєднання підпорядковано принципам незмінності об’єктів.
2. Деструктивну. Для кожного піддерева під час побудови будуються відповідні списки неініціалізованих нетерміналів, що знаходять своє продовження в наступному піддереві. Тоді у зворотному порядку проводиться присвоєння адрес в неініціалізовані вузли, визначивши таким чином поєднане дерево розбору.

Конструктивна стратегія має сенс у випадках, коли необхідно зберігати часткові дерева розбору після побудови цілісного. Але зважаючи на відсутність необхідності такого результату, то з метою явної оптимізації можна використати деструктивний підхід, що може значно заощадити ресурси на поєднанні. Оскільки швидкодія в даній задачі є критичним фактором, то в проєкті використовується деструктивний підхід для поєднання.

2.2.4 Ієрархічна специфікація мови

Наведемо поняття, введені в [7] для обґрунтування подальшої побудови алгоритмів у проєкті.

Ієрархічна специфікація мови (ІСМ) – це пара (*P*, *V*), де *P* – множина пар $(P',\ M)$, де в свою чергу P' – PF(*)-таблиці для специфікації мови, що обмежується парою символів-маркерів M. S, для якого справедливо $S \in P$ – початкова пара символів-маркерів.

Покажемо поняття пари символів-маркерів та пов'язаних з ними часткових специфікацій мови. Для цього розширимо мову Дика до наступної:

$$G_r = (N_r,\ T,\ P,\ S), \tag{9}$$

де множина термінальних символів задається формулою 10:

$$T = \{\ a_1,\ a_2,\ ...,\ a_r\ \} \cup \{\ \overline{a_1},\ \overline{a_2},\ ...,\ \overline{a_r}\ \} \cup \bigcup_{1 \le i \le r} \Delta_i, \tag{10}$$

де $\Delta_i = \bigcup_{1 \le i \le r} \{\ c_{i_1}, c_{i_2}, ...,\ c_{i_{n_i}}\ \}$;

множина нетерміналів задається формулою 11:

$$N_r = T \cup \{S\} \cup C, \tag{11}$$

де $C = \bigcup_{1 \le i \le r} \left\{ C_{i_1}, C_{i_2}, ..., C_{i_{n_i}} \right\}$.

Обмежимо цю мову лише правилами вигляду:

$$S \to a_i \; S \; C_{i_1} C_{i_2} .... \; C_{i_{n_i}} \overline{a_i} \; S \; | \; \varepsilon, \tag{12}$$

де $C_{i_j} \to c_{i_j} S \mid \varepsilon$, $1 \le i \le r$ і $1 \le j \le n_i$

Тоді пара $\left(a_i, \; \overline{a_i} \right)$ є парою символів-маркерів *M*.

Для специфікації P' вводиться функція, що відображає властивості взаємних зв'язків між нетерміналами.

$$\mu(\varepsilon) = \varepsilon,$$

$$\mu(xa_i) = \mu(x)a_i \text{ для } 1 \le i \le r,$$

$$\mu(xc_i) = \mu(x)c_{i_j} \text{ для } 1 \le i \le r \text{ і } 1 \le j \le n_i,$$

$$\mu(x\overline{a_i}) = \begin{cases} \mu(x)\overline{a_i}, & якщо \quad \mu(x) \notin T * a_i y \\ x', & якщо \quad \mu(x) = x' a_i y \end{cases} \text{ для } 1 \le i \le r,$$

де y – рядок, що генерується граматикою нижче:

$$G_i = \left(N_i, \; T_i, \; P_i, \; S_i \right),$$

де множина нетерміналів $N = \left\{ C'_{i_1}, C'_{i_2}, ..., \; C'_{i_{n_i}} \right\} \cup \left\{ Y_i \right\}$. Ця граматика складається лише з правил наступного вигляду, відображеному формулою 13.

$$Y_i \rightarrow C'_{i_1} C'_{i_2} .... C'_{i_{n_i}}, \quad (13)$$

де нетермінали задані наступним чином $C'_{i_j} \rightarrow c_{i_j} S \mid \varepsilon$.

Також вагомим елементом, необхідним для використання ієрархічної специфікації мови є граф залежності маркерних пар (ГЗМП). Цей граф визначає ієрархічну залежність маркерних пар із ієрархічної специфікації мови.

Нехай *(a, b)* – маркерна пара із певної $p \in P$ із ІСМ. Така пара може використовуватись як термінальні символи підмови для іншої пари $q \in P$ з множиною терміналів *T*. Тоді нехай *x* – рядок, що генерується підмовою *q* і для нього виконується рівність:

$$x = uav,$$

де $u, v \in T^*$. Тоді *v* має відповідати наступній умові:

$$v = by,$$

де $y \in T^*$. Таким чином забезпечується обов'язкова умова слідування *a* та *b*. Аналогічно для $x = ubv$, має виконуватись $u = za$, де $z \in T^*$.

Тоді вводиться відношення *використання* між маркерною парою (a_1, b_1) із $p \in P$ та іншою парою (a_2, b_2) із $q \in P$. ГЗМП фіксує цей зв'язок у вигляді графа. Тому вершинами (*V*) цього графа є всі маркерні пари, а дуги (*E*) підлягають наступному формулюванню:

$$E = \left\{ v_1 \rightarrow v_2 \mid v_1 \quad використ. \quad v_2,\ v_1,\ v_2 \in V \right\} \quad (14)$$

Таким чином комбінація ІСМ та ГЗМП формалізує властивості мови, задані БНФ у форму, зручну для реалізації в паралельному алгоритмі синтаксичного аналізу.

2.3 Обробка помилкових ситуацій

Помилкові ситуації (знаходження невідповідності переданого рядка мові) можуть ставатися на таких стадіях:

1 Незалежно від взаємних зв’язків між нетерміналами (на етапі лексичного аналізу – помилка «непідтримуваний символ»)
2 На стадії побудови часткового дерева розбору (неможливість побудувати часткове дерево розбору виходячи із заданих передбачень щодо стану попереднього фрагменту)
3 На стадії поєднання часткових дерев розбору в цілісне (неможливість поєднання часткового дерева розбору, щоб результуюче цілісне дерево розбору відповідало вихідній мові)

3 РОЗРОБКА КОМПОНЕНТІВ СИСТЕМИ

3.1 Огляд використаних інструментів

3.1.1 Мова програмування Common Lisp

LISP (LISt Processing) – сімейство мов програмування загального призначення з підтримкою парадигм функціонального, процедурного та об’єктно-орієнтованого програмування.

Мову програмування Lisp було розроблено у Масачусетському Технологічному Інституті для дослідження задач, що були пов’язані зі штучним інтелектом в кінці 1950-тих. Поняття штучного інтелекту та пов’язані з ним задачі, з тих пір кардинально змінились, але через простоту та потужність закладених принципів, мову можна застосовувати і в багатьох інших сферах.

Common Lisp – діалект мови програмування Lisp і специфікує всі існуючі діалекти Lisp так, що нині найвідоміші компілятори Lisp підпорядковуються саме стандарту ANSI щодо Common Lisp [17].

Мова підтримує наступні парадигми програмування:

- Процедурну (можливість декларування іменованих функцій за допомогою defun).
- Об’єктно-орієнтовану (спеціальна об’єктно-орієнтована система CLOS – Common Lisp Object System з нетиповими можливостями декларування методів поза класами та метод-комбінаторами).
- Функціональну (наявність можливості створення лямбда-виразів як неіменованих функцій, та наявність типових функціональних примітивів на кшталт map, reduce та ін.)

Також існують можливості із застосування метапрограмування – техніка програмування, коли комп’ютерна програма має можливість використовувати інші програми як дані. Ці можливості широко реалізовуються в мові за допомогою макросів, функцій read, eval, print, quote, що забезпечують повний

контроль над програмним кодом на етапі компіляції, що дозволяє його видозміну та трансформації.

3.2 Композиція ЛА та СА

Підходи до інтеграції ЛА і СА при розробці компіляторів можна підрозділити на наступні:

- однопрохідний аналізатор, при роботі якого синтаксичний аналізатор компонує в собі властивості одночасно як ЛА, так і СА таким чином, що СА «на льоту» із послідовності символів зчитує спочатку лексему, а вже потім передає її для подальшого розбору нетермінала;
- двопрохідний аналізатор, що спочатку перетворює послідовність символів на заданій мові у послідовність лексем (перший прохід) і вже із послідовності лексем будує результат роботи СА – дерево розбору.

Головною перевагою однопрохідного і одночасно недоліком двопрохідного аналізатора є швидкодія першого. Це досягається за рахунок відсутності в однопрохідного аналізатора стадії створення послідовності лексем, що б в результаті забезпечило ріст затраченого часу та пам’яті на створення і збереження всього рядка лексем. Звісно можна під час другої стадії – СА забезпечити звільнення пам’яті під час зчитування рядка лексем, щоправда такий підхід не вбереже від втрати швидкодії. Тому саме перший підхід застосовується в переважній більшості сучасних САГСА.

З іншого боку, якщо програміст реалізовує СА за допомогою лише мови програмування без спеціалізованих бібліотек, парсер-комбінаторів або генераторів парсера із БНФ, то без сумніву, підхід розділення ЛА та СА, незважаючи на свою непродуктивність з боку виконуваного коду, вирізняється своєю продуктивністю розробки, оскільки програмісту не доводиться вручну прописувати дублювання коду ЛА для обробки кожної лексеми в СА, що

пришвидшує швидкість розробки, зменшує кількість створеного програмного коду, а отже і зменшує кількість можливих допущених помилок.

З точки зору міркувань, наведених в попередньому розділі можна зробити висновок, що логічне поєднання ЛА та СА призведе до зростання кількості неоднозначності при розборі частин заданої послідовності символів, що неодмінно призведе до падіння швидкодії. В наступному розділі це твердження буде піддано експериментальній перевірці.

3.3 Структура програмного комплексу

Розроблений програмний комплекс може бути розділений на такі компоненти, що відповідають за:

- обробку вхідної граматики;
- побудову послідовного лексичного та синтаксичного аналізаторів;
- побудову паралельного лексичного та синтаксичного аналізаторів;
- побудову генератора тестових прикладів для вхідної граматики;
- перевірку побудованих дерев розбору;
- демонстрацію швидкодії всіх алгоритмів та політик їх використання.

В додатку 2 представлена структура модульних взаємозв’язків системи.

3.3.1 Модуль обробки вхідної граматики

Даний модуль відповідає за конвертацію представлення мови з s-виразів у внутрішнє представлення (хеш-таблиці) граматики для подальшого використання в послідуючих модулях, що її використовують.

Граматика представляється мовою s-виразів, несучи подібне до БНФ семантичне навантаження.

Для використання названої функціональності було створено макрос with-grammar. З його допомогою можна декларувати правила граматики на зразок макросу let. Тобто кожне правило представляється s-виразом, перший елемент якого є ідентифікатором правила, а решта є переліком списків терміналів та нетерміналів, які в БНФ проставлялись через операцію «або», тобто цей перелік визначає альтернативи розгортки правила.

В кожній з таких альтернатив є список терміналів та нетерміналів, що характеризують розбір правила.

3.3.2 Лексичний і синтаксичний аналізатори

Термінали мови задаються показаним на рисунку нижче чином.

```
(defun demo (input)
  (with-grammar ((term (char)
                       (char term))
                 (char ("0")))
    (parse input :as term)))
```

Рисунок 7 – Базовий приклад використання створеного СА

Поділ на лексичний та синтаксичний аналізатори виконається автоматично, якщо не буде задано відповідні нетермінали для утворення лексичного аналізатора з допомого спеціального ключового слова “:token” правила. На рисунку нижче подано приклад такого декларування.

```
(defun demo (input)
  (with-grammar ((term (char)
                       (char term))
                 (:token char ("0")))
    (parse input :as term)))
```

Рисунок 8 – Граматика з явно заданим лексичним правилом

Також для перевірки символу можна передати предикат. Цей предикат повинен бути чистою функцією [14], тобто не повинен виконувати так званих побічних ефектів і завжди на одні й ті ж вхідні дані повинен бути той самий відповідний результат.

```
(defun parse-digits (input)
  (with-grammar ((term (digit)
                       (digit term))
                 (:token digit ((:predicate #'digit-char-p))))
    (parse input :as term)))
```

Рисунок 9 – Використання предикату в правилі граматики

Для об’єктно-орієнтованого представлення терміналів граматики з метою застосування поліморфізму побудована ієрархія класів, що представлена в додатку 3.

Для реалізації послідовного синтаксичного аналізатора було побудовано низхідний генератор LL(*) CA – машину Кнута [15].

Для реалізації паралельного синтаксичного аналізатора побудовано алгоритм, що за допомогою ієрархічної специфікації мови та графа залежності маркерних пар.

На вхід даний алгоритм приймає на вхід БНФ представлення граматики мови тому має сенс виконати початкову побудову ІСМ та ГЗМП для подальшого багаторазового використання із різними вхідними рядками.

Далі наведено псевдокод алгоритму побудови ІСМ та ГЗМП із граматики, поданої в БНФ.

Вхідні дані: граматика в БНФ.

Початок.

1. Пов’язати нетермінальні символи за критерієм розгортки одного в інший.

2. Відповідно до методу, наведеному в підпункті 2.2.4 визначити набір унікальних маркерних пар.

2.1. Набір нетерміналів, що розгортаються між знайденими маркерними парами вважати окремою регулярною мовою і побудувати для неї PF(1)-таблиці.

2.1.1. Якщо відповідно до PF(1)-таблиці не всі термінали підлягають розпізнаванню за критерієм з формули 8 з підпункту 2.2.2, то необхідно збільшити кількість символів для перегляду і побудувати таблицю множин PF(k), де k – лічильник, що послідовно зростає на 1, поки для всіх терміналів не почне виконуватись критерій з формули 8.

2.2. На основі зв’язків терміналів, визначити за відношеннями використання з формули 14, побудувати граф залежності маркерних пар.

3. Виконати топологічне сортування ГЗМП.

4. Зкомпонувати за парами маркерних символів залежні маркерні пари та таблиці множин PF(k).

Кінець.

Тепер наведемо псевдокод алгоритму паралельного лексичного аналізу основі PF(k).

Вхідні дані:

- рядок для розбору *str*,
- кількість паралельних аналізаторів *n*,
- таблиці множин PF(k) для лексичного аналізатора *lexer-pfs*,

Початок.

1. Розділити *str* на n рівних за розміром пам’яті фрагменти.

1.1. Для кожного фрагмента виконаємо зсув початку фрагмента на початок символу у випадку, якщо символ закодовано кодом зі змінною довжиною.

2. Запустити перший ЛА на першому фрагменті з класичним алгоритмом послідовного ЛА.

2.1. По закінченню розбору очікувати завершення решти аналізаторів, очікуючи при цьому можливе зміщення правого краю послідовності.

2.2. Провести розбір доданої зміщеної частини і поєднати отриманий результат.

3. Паралельно з першим, запустити ($n - 1$) алгоритмів паралельного ЛА, використовуючи PF(k)-таблиці для лексичного розбору.

3.1. Для кожного з потоків виконання:

3.1.1. Провести зсув до початку лексеми.

3.1.2. Провести аналіз фрагмента за таблицею *lexer-pfs* з побудовою відповідного часткового списку лексем.

4. Поєднуємо часткові результати в список лексем.

Кінець.

Тепер наведемо псевдокод алгоритму паралельного СА за ІСМ та ГЗМП.

Вхідні дані:

- рядок для розбору *str*,
- кількість паралельних аналізаторів *n*,
- PF(k)-таблиці для синтаксичного аналізатора *parser-pfs*,
- ГЗМП *graph*.

Початок.

1. Розділити вхідний рядок на n рівних за кількістю лексем фрагменти.

1.1. Запустити перший ЛА на першому фрагменті з класичним алгоритмом послідовного СА.

1.2. По закінченню розбору очікувати завершення решти аналізаторів, очікуючи при цьому можливе зміщення правого краю послідовності.

1.3. Провести розбір доданої зміщеної частини і поєднати отриманий результат.

1.4. Паралельно з першим, запустити алгоритм паралельного ЛА, використовуючи *parser-pfs* та *graph*.

1.4.1. Лінійним пошуком з початку фрагмента str знаходимо маркер-символ початку нетермінала.

1.4.2. Використовуючи *parser-pfs* та *graph* отримуємо часткове дерево розбору.

2. Поєднуємо часткові результати в єдине дерево розбору на основі *graph*.

Кінець.

Для використання послідовного СА, необхідно викликати функцію parse із ключовим параметром ":worker-count" зі значенням 1, як зображено на рисунку нижче, або не задавати параметр ":worker-count" взагалі, оскільки значення 1 встановлюється за замовчуванням.

```
(defun parse-digits (input)
  (with-grammar ((term (char)
                       (char term))
                 (:token char (:predicate #'digit-char-p)))
    (parse input :as term
                 :worker-count 1)))
```

Рисунок 10 – Використання послідовного СА

Для використання відповідного паралельного СА необхідно задати той же параметр, але зі значенням, що більше за 1, вказуючи таким чином кількість паралельних обробників, як зображено на рисунку нижче.

```
(defun parse-digits (input)
  (with-grammar ((term (char)
                       (char term))
                 (:token char (:predicate #'digit-char-p)))
    (parse input :as term
                 :worker-count 4)))
```

Рисунок 11– Використання паралельного СА

Таким чином створено умови для переключання між режимами запуску СА. Ці можливості також, як і правила нетерміналів, набули відображення в ієрархії успадкування класів, що зображені на додатку 3.

3.3.3 Модуль генерації речення заданої мови

Макровизначення with-grammar також дозволяє в подібно до аналітичної задавати також і генеруючу граматику. Зміни набуває лишу функція, яку треба викликати для генерації речення – generate. Вона приймає за параметр термінал із прив'язок макровизначення with-grammar.

Дане представлення граматики дозволяє вносити прямо в правило спеціальні позначення, що відповідають за внесення додаткових символів у результуючу послідовність символів.

Зокрема такий підхід можна застосовувати для моделювання конвенцій з дотримання певної кількості пробілів, табуляцій та інших терміналів, що не впливають на саме по собі результуюче дерево розбору, але можуть вплинути

на швидкодію. Тому не варто ігнорувати такі властивості мов і заносити їх в генеруючу властивість граматики. Крім того, таке доповнення дозволяє візуально краще сприймати згенеровані рядки. Для прикладу граматику генерації цифр було доповнено генерацією пробілів між цифрами з допомогою ключового слова “:insert” на рисунку нижче.

```
(defun generate-digits (count)
  (with-grammar ((term (digit)
                       (digit
                        (:insert " ")
                        (:limit term count)))
                 (:token digit ((:predicate #'digit-char-p
                                 :assign #\9))))
    (generate term)))
```

Рисунок 12 – Доповнення генеруючої граматики

Іншою важливою особливістю даного модуля є можливість надання обмеження кількості розгортань рекурсивних правил. Очевидно, що рекурсивні правила, що задають можливості синтаксичного аналізу, при їх переносі в застосування для генерації вже вхідних послідовностей дається взнаки їх необмеженість. Тому для кожного рекурсивного терміналу для побудови текстових вхідних прикладів необхідно вказати числові обмеження їхньої кількості.

```
(defun generate-digits (count)
  (with-grammar ((term (digit)
                       (digit (:limit term count)))
                 (:token digit ((:predicate #'digit-char-p
                                 :assign #\9))))
    (generate term)))
```

Рисунок 13 – Генеруюча граматика з простим “:assign”

В доповнення до останньої опції, можна надати визначене користувачем значення для термінала за допомогою ключового слова “:assign”. Для заданого термінала або нетермінала проводиться синтаксичний аналіз вказаного користувачем значення з метою перевірки, чи не порушує воно граматики мови (див. рисунок нижче).

```
(defun generate-digits ()
  (with-grammar ((top ((term
                        :assign "000000")))
                 (term (digit)
                       (digit term))
                 (:token digit ((:predicate #'digit-char-p))))
    (generate top)))
```

Рисунок 14 – Генерація з перевіркою значення “:assign”

3.4 Поділ послідовності символів у випадку кодування UTF-8

Кодування UTF-8 характеризується змінною довжиною символів, що провокує складності у правильному поділі послідовності символів лише за розміром частин. Так виникає можливість розділення символу і потрапляння його частин у різні сегменти програмного коду, що розділялися з метою їх передачі до паралельних СА. В даному підрозділі пропонується спосіб за константний час змістити початок зчитування для сегмента вихідної послідовності, щоб уникнути проблем з подальшими етапами аналізу.

Таблиця 2 – Представлення Unicode символів в кодуванні UTF-8

‘	UTF-8
0x00000000 — 0x0000007F	0xxxxxxx
0x00000080 — 0x000007FF	110xxxxx 10xxxxxx

0x00000800 — 0x0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
0x00010000 — 0x001FFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Отже, зважаючи на те, що октети, старші біти яких дорівнюють 10 (в двійковій системі числення), то можна виконати лінійний пошук з метою знаходження початку символу в кодуванні UTF-8 [11]. Для цього, октет, що починається не з 10 можна вважати початком символу і зміщувати на цю знайдену адресу початок фрагмента для часткового паралельного аналізатора.

Подібним чином можна опрацьовувати інше змінне кодування – UTF-16, розмір символів якого може бути або 2, або 4 байти.

3.5 Поєднання часткових дерев розбору з точки зору ОС

Часткові дерева розбору знаходяться в пам’яті різних потоків програми СА, що вимагає передачі пам’яті цих новостворених потоків під управління головного потоку, що їх породив. Для реалізації цих можливостей компілятор SBCL у своїй надбудові над системними примітивами багатопоточності та синхронізації ОС Linux надає можливості з повернення потоком результату без накладних затрат.

4 ТЕСТУВАННЯ СИСТЕМИ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

4.1 Характеристика системи тестування

Алгоритм системи тестування ілюструється в додатку 4.

Враховуючи складність та можливу нестабільність побудованого паралельного алгоритму СА, необхідно створити систему, що б забезпечувала тестування побудови часткових дерев розбору, і, відповідно, вже з них перевіряла результуюче дерево розбору на правильність. Отже, необхідно використати модуль, що б генерував речення заданої мови з регульованим різноманіттям та обсягом результуючого речення. Також необхідне джерело дерева розбору для перевірки побудованого паралельним СА.

Крім перевірки дерева розбору також важливим є наявність інструментів для порівняння швидкодії та затрачених ресурсів відповідних послідовного та паралельного СА. Для цього було додано модуль побудови послідовного СА із тої самої граматики, що і для паралельного. Таким чином також розв’язується проблема перевірки дерева розбору на відповідність, оскільки дерево розбору, створене послідовним СА вважається апріорно правильним. Це так, звичайно, за умови повного покриття всіх частин послідовного СА тестами, що було забезпечено.

Для вирішення викладеної вище задачі використано реалізовані в проєкті модулі генерації рядка за граматикою, генерації відповідного граматиці послідовного та паралельного ЛА та СА. Тому такий тест виглядає наступним чином, показаним на рисунку нижче.

```
(deftest test-parse-digits
    (testing "parallel parse validity"
             (with-grammar ((term (char)
                                  (char (:limit term 10)))
                            (:token char (:predicate #'digit-char-p)))
               (let ((input (generate term)))
                 (ok (equalp (parse input :as term
                                          :worker-count 1)
                             (parse input :as term
                                          :worker-count 4)))))))
```

Рисунок 15– Приклад декларування тесту

Для побудови і систематизації тестів було використано тестовий фреймворк rove [16], що забезпечує реалізацію використаних макросів deftest, testing та ok.

4.2 Порівняння продуктивності паралельного CA відносно послідовного на граматиках

4.2.1 Характеристика оточення для проведення тестування

Для кожного виміру виконаємо 30 тестових запусків з метою збільшення точності часового результату. Що ж до апаратного забезпечення, то для послідуючих вимірів було використано процесор з 6-ма фізичними процесорами Intel Core i5-8400.

4.2.2 Порівняння для підмножини мови C

Мова, використана для тестування, включає в себе деяку підмножину синтаксичних властивостей мови C, зокрема частини, що відповідають за пріоритетність виразів, основні керуючі структури та оголошення змінних та

функції. Докладніше ознайомитись із граматикою мови, використаної для тестування можна в додатку з лістингом розробленої програми.

Порівняємо час виконання послідовного та паралельного ЛА підмножини мови С.

Таблиця 3 – Порівняння часу виконання послідовного та паралельного ЛА підмножини мови С

Розмір вхідної послідовності, КБ	Час виконання послідовного		Прискорення при використанні паралельних ЛА		
	ЛА, мс	СА, мс	2	4	6
16	3,58	10,68	0,78	0,42	0,39
180	16,12	30,24	1,45	1,32	1,24
480	45,64	91,25	0,89	1,17	2,23
9244	583,45	1511,63	1,48	2,28	3,23
62480	3485,08	9390,82	1,86	2,58	3,64

Враховуючи дані, отримані із тестових замірів часу, можна зробити висновок, що ріст довжини вхідної послідовності провокує ріст прискорення паралельного ЛА відносно послідовного.

Також можна дійти висновку, що для даної мови ресурси, затрачені на задачу ЛА порівнювані з відповідними затратами на СА, перебуваючи в діапазоні 25-35% від загального затраченого часу на повний послідовний запуск ЛА та СА, є порівнюваними і не є нехтовними. Тому необхідно дослідити вплив паралелізації ЛА на швидкодію композитного аналізатора, що складається з незалежних і паралельно запущених аналізаторів, що в свою чергу складаються з послідовно запущених часткових ЛА та СА.

Порівняємо дві політики поєднання ЛА та СА: паралельний СА при послідовному ЛА та паралельний СА на частковому списку лексем, створених кожним незалежним паралельно запущеним ЛА.

Таблиця 4 – Порівняння прискорення паралелізації СА в залежності від паралелізації ЛА підмножини мови С

Розмір вхідної послідовності, КБ	Прискорення при використанні послідовного ЛА			Прискорення при використанні паралельного ЛА		
	2	4	6	2	4	6
16	1,35	1,13	0,64	1,15	0,72	0,47
180	2,18	1,85	1,58	1,45	1,79	1,47
480	1,28	1,65	1,55	1,22	2,05	2,42
9244	1,37	2,15	1,97	1,32	2,16	2,04
62480	1,76	2,65	3,26	1,61	2,77	3,71

Враховуючи отримані дані, можна дійти висновку, що очевидної переваги паралелізації ЛА не виявлено для розмірів файлів, використання яких мало б практичний інтерес. Це можна пояснити відносною складністю ЛА для вказаної мови (враховуючи численні неоднозначності, що виникають при намаганні застосувати паралельний ЛА (див. підпункт 2.1.2).

4.2.3 Порівняння для мови JSON

Проведемо порівняння прирости швидкодії ЛА з використанням різної кількості використаних потоків.

Таблиця 5 – Порівняння часу виконання послідовного та паралельного ЛА мови JSON

Розмір вхідної послідовності, КБ	Час виконання послідовного		Прискорення при використанні паралельних ЛА		
	ЛА, мс	СА, мс	2	4	6
2,4	0,309	0,755	0,77	0,81	0,62
24	1,989	3,327	1,15	1,22	1,14
140	18,45769	31,28803	0,41	0,92	0,84
8644	362,894	552,23	1,46	1,52	1,44
74640	3426,227	5348,399	1,52	2,32	2,84

Аналогічно з випадком для С, спостерігається певний ріст (хоча і обмежений) прискорення ЛА внаслідок зростання обсягу задачі розбору.

Таблиця 6 – Порівняння прискорення паралелізації СА в залежності від паралелізації ЛА мови JSON

Розмір вхідної послідовності, КБ	Прискорення СА при використанні послідовного ЛА			Прискорення СА при використанні паралельного ЛА		
	2	4	6	2	4	6
24	1,67	1,42	1,23	2,02	1,91	1,73
240	1,84	2,48	2,15	2,42	2,51	2,28
640	1,43	2,94	2,62	1,28	1,34	1,96
8644	1,75	2,86	2,88	2,12	2,87	2,99
74640	1,82	2,97	3,25	2,34	3,26	3,96

Варто зазначити, що приріст швидкодії порівняно низький, коли використано неповну кількість фізичних процесорів, то можлива міграція потоку виконання із одного процесора на інший, що негативно впливає на

швидкодію. Це можна пояснити, оскільки міграція пам’яті потоку може провокує значне пониження швидкодії, оскільки значення не буде наявне в кеші. З іншого боку, повне навантаження процесорів унеможливлює виникнення вказаної ситуації.

Також відповідно до отриманих даних, помітний приріст швидкодії із використання паралельного ЛА можна помітити лише в останнього тестового запуску і з зростанням розміру вхідного файлу цей приріст очікується до зростання.

ВИСНОВКИ

Розроблений програмний комплекс дозволяє автоматично будувати та тестувати паралельний СА за наведеною в проєкті формою подання БНФ в формі s-виразів. Для цього було застосовано:

- загальні теоретичні напрацювання в теорії формальних граматик;
- спеціалізовані методики побудови паралельного синтаксичного аналізу (ієрархічна специфікація мови, граф залежностей маркерних символів);
- мова Common Lisp та компілятор SBCL;
- стандарти специфікації використаних для тестування швидкодії паралельного СА мов,
- довідкові дані щодо Unicode кодувань, нюансів роботи архітектури з кеш-пам’яттю та дані щодо використання бібліотек для Common Lisp.

Використання паралельного СА може не бути ефективним, якщо розмір послідовності для розбору порівняно малий. Основні проблеми для застосування паралельного СА для послідовності, розмір якої менше за певне специфічне для мови значення:

- супутні затрати на запуск та синхронізацію паралельних аналізаторів, що при використанні паралельного СА на невеликих розмірах задачі не надто пришвидшать аналіз, а можливо такий підхід покаже навіть меншу ефективність, ніж відповідний послідовний СА;
- вагома частина часу виконання послідовно виконуваного коду у порівнянні з паралельною (поєднання часткових дерев розбору стає порівнюваною операцією з частковим СА).

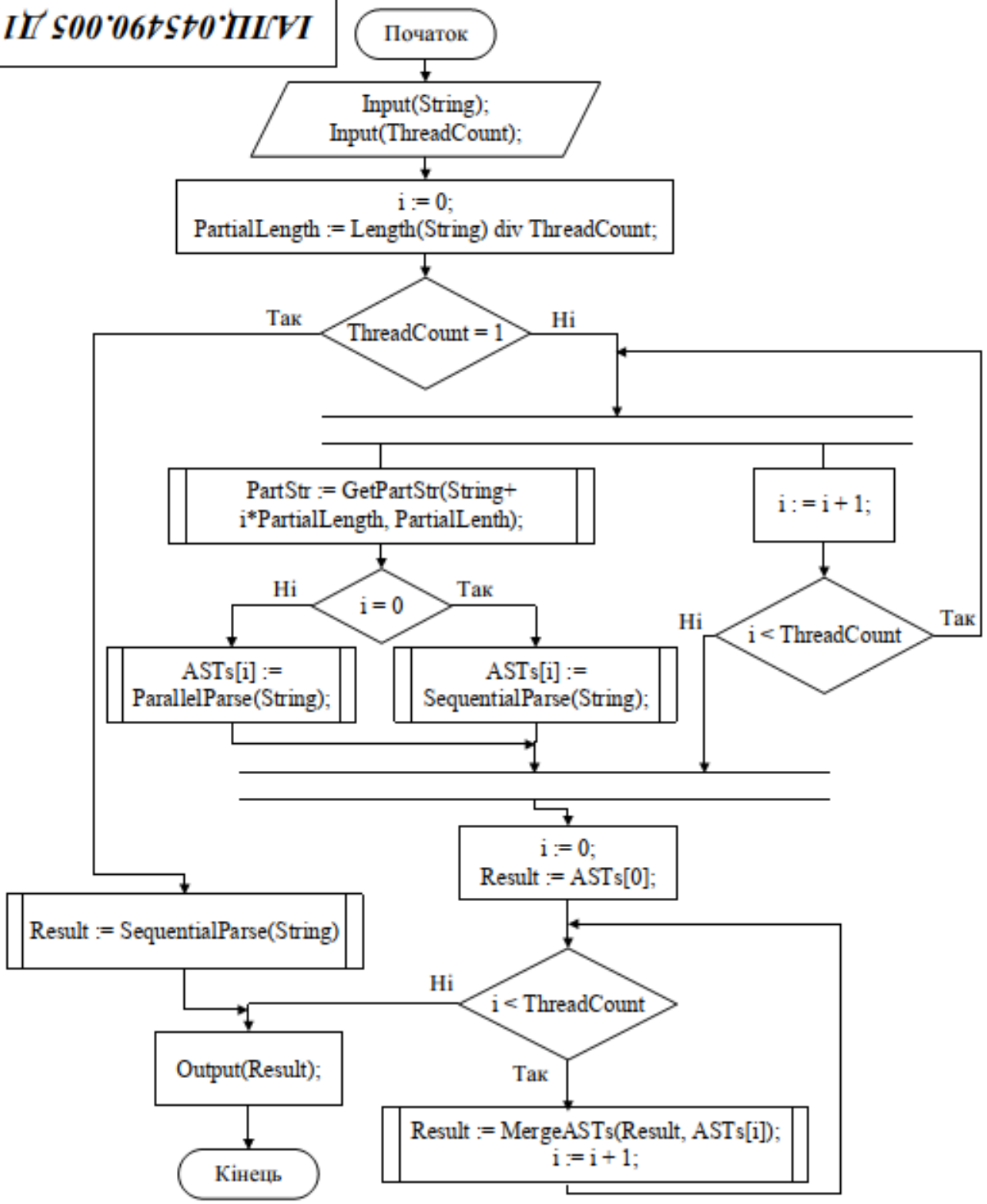
Оскільки час, затрачений на послідовне виконання відіграє роль знижувача потенціалу паралелізації СА, то для мов з простішим синтаксисом, ефективність використання паралельного СА буде вище, ніж для мов із складнішим синтаксисом.

Відповідно до отриманих даних в розділі 4, можна дійти висновку, що паралелізація ЛА є важливою. Передумови для цього твердження створює порівняння затраченого часу на відповідні для досліджуваних мов ЛА та СА. Виходячи з отриманих даних це 25-35% повного сумарного запуску ЛА та СА. Тому забезпечення паралельного виконання лексичного аналізу важливо. Також важливим є спосіб композиції паралельних ЛА та СА. Зокрема, з боку використання кеш-пам’яті може бути ефективним забезпечення виконання часткового синтаксичного аналізу на частковому списку лексем. Таким чином шанс збереження часткового спуску лексем у кеш-пам’яті є вищим, ніж якби всі часткові результати ЛА поєднувалися в одне ціле, а потім поділялись між паралельними СА. Крім того, це дозволяє зекономити на зайвому поєднанню та запуску потоків, хоча при цьому необхідно забезпечити механізм синхронізації при зсуві початку в частковій послідовності лексем для часткового СА та переносі відповідних лексем під розбір попереднього аналізатора.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 1956. — T. Vol.2. — C. 113–124.
2. Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 1959. — T. Vol.2. — C. 137–167.
3. S. Doaitse Swierstra. Combinator parsers: From toys to tools. *Electronic Notes in Theoretical Computer Science*, 2001. №41, C. 38–59
4. Bryan Ford. Parsing Expression Grammars: A Recognition Based Syntactic Foundation. *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2004. C. 111 – 115.
5. Johnson, Stephen C. Yacc: Yet Another Compiler-Compiler (Technical report). URL: http://dinosaur.compilertools.net/yacc/ (дата звернення: 26.04.2020).
6. R. M. Schell, Jr., Methods for Constructing Parallel Compilers for Use in a Multiprocessor Environment. Ph.D. Thesis, Computer Science Dept., Univ. of Illinois, Urbana, 1979
7. Y. N. Srikant, Priti Shankar. Parallel Parsing of Programming Languages. *Information Sciences*, 1987. №43, C. 55 – 83.
8. C. N. Fischer, On Parsing Context-Free Languages in Parallel Environments, Ph.D. Thesis, Cornell Univ., 1975.
9. Here-Document description in the POSIX/SUS standard. URL: https://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html (дата звернення: 26.04.2020).
10. The Parallel GCC. URL: https://gcc.gnu.org/wiki/ParallelGcc (дата звернення: 21.04.2020).
11. The Unicode Standard Version 13.0 – Core Specification. General Structure. URL: https://www.unicode.org/versions/Unicode13.0.0/ch02.pdf (дата звернення: 10.05.2020).

12. Пентус А. Е., Пентус М. Р. Теория формальных языков: Учебное пособие. – М.: Изд-во ЦПИ при механико-математическом ф-те МГУ, 2004. – С. 80.
13. Amdahl, G. The validity of the single processor approach to achieving large-scale computing capabilities. *In Proceedings of AFIPS Spring Joint Computer Conference, Atlantic City, N.J., AFIPS Press*, 1967. С. 483-485.
14. Brian Lonsdorf (2015). Professor Frisby's Mostly Adequate Guide to Functional Programming. URL: https://github.com/MostlyAdequate/mostly-adequate-guide/blob/master/ch03.md (дата звернення: 21.04.2020).
15. D. E. Knuth. Top-down syntax analysis. *Acta Inf.*, 1971. С. 79–110.
16. The rove Reference Manual. URL: https://quickref.common-lisp.net/rove.html (дата звернення: 28.04.2020).
17. CLHS: About the Common Lisp HyperSpec (TM)". URL: http://www.lispworks.com/documentation/HyperSpec (дата звернення: 21.04.2020).



Підп. і дата | Інв. № дубл. | Взам. інв. № | Підп. і дата | № подл.

Зм	Лист	№ докум.	Підпис	Дата
Розроб.		Вовчок О. В.		
Перевірив		Зорін Ю. М.		
Т. Контр.				
Н. Контр.		Клятченко Я.М.		
Затвердив		Романкевич В.О.		

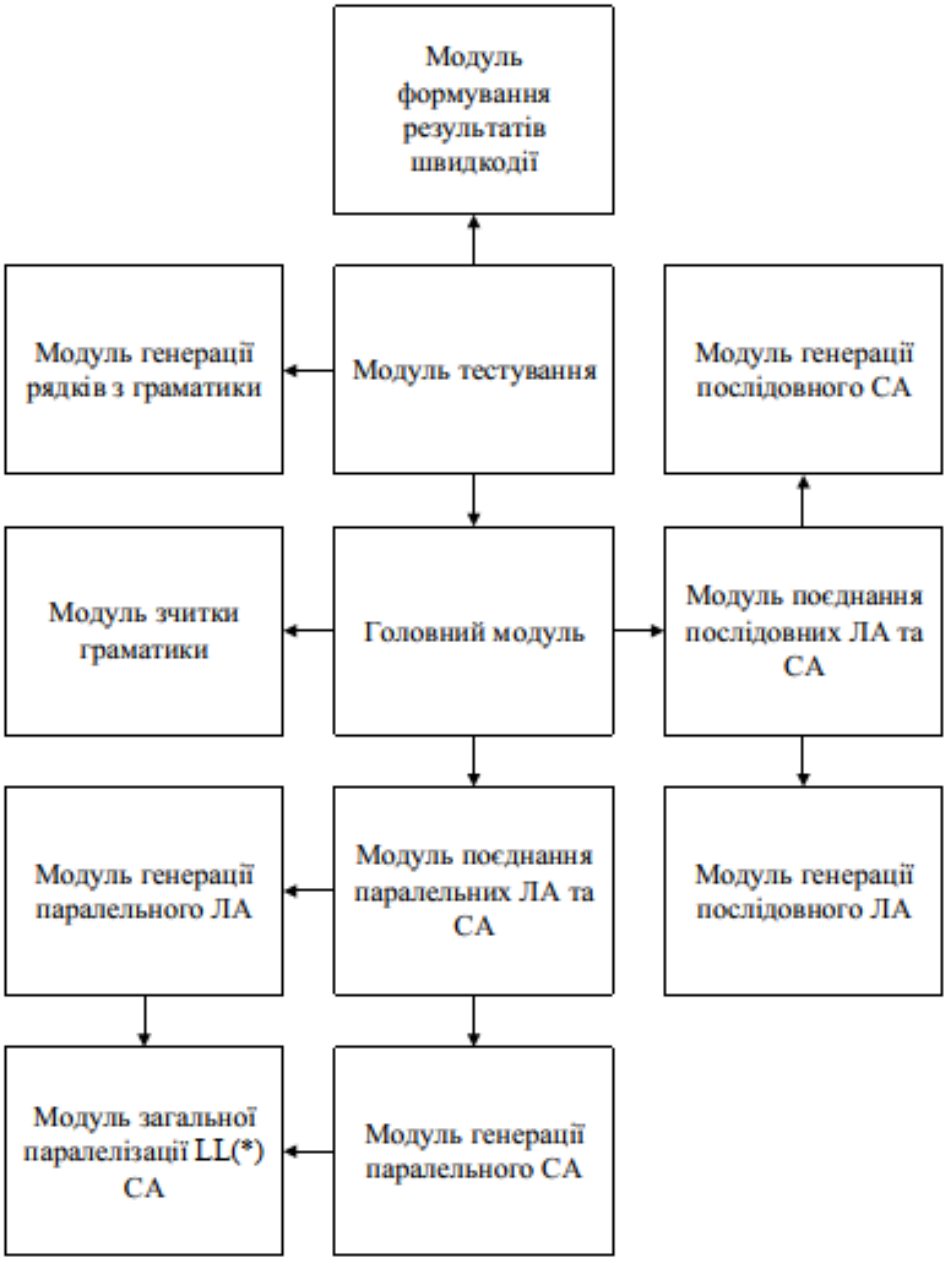
ІАЛЦ.045490.005 Д1

Алгоритм паралельного синтаксичного аналізатора. Схема алгоритму

Літ.	Маса	Масштаб
Аркуш 1	Аркушів 1	

Паралельно розподілений синтаксичний аналізатор

КПІ ім. Ігоря Сікорського, ФПМ, КВ-62



Підп. і дата | Інв. № дубл. | Взам. інв. № | Подп. И дата | № подл.

Зм	Лист	№ докум.	Підпис	Дата
Розроб.		Вовчок О. В.		
Перевірив		Зорія Ю. М.		
Т. Контр.				
Н. Контр.		Клятченко Я.М.		
Затвердив		Романкевич В.О.		

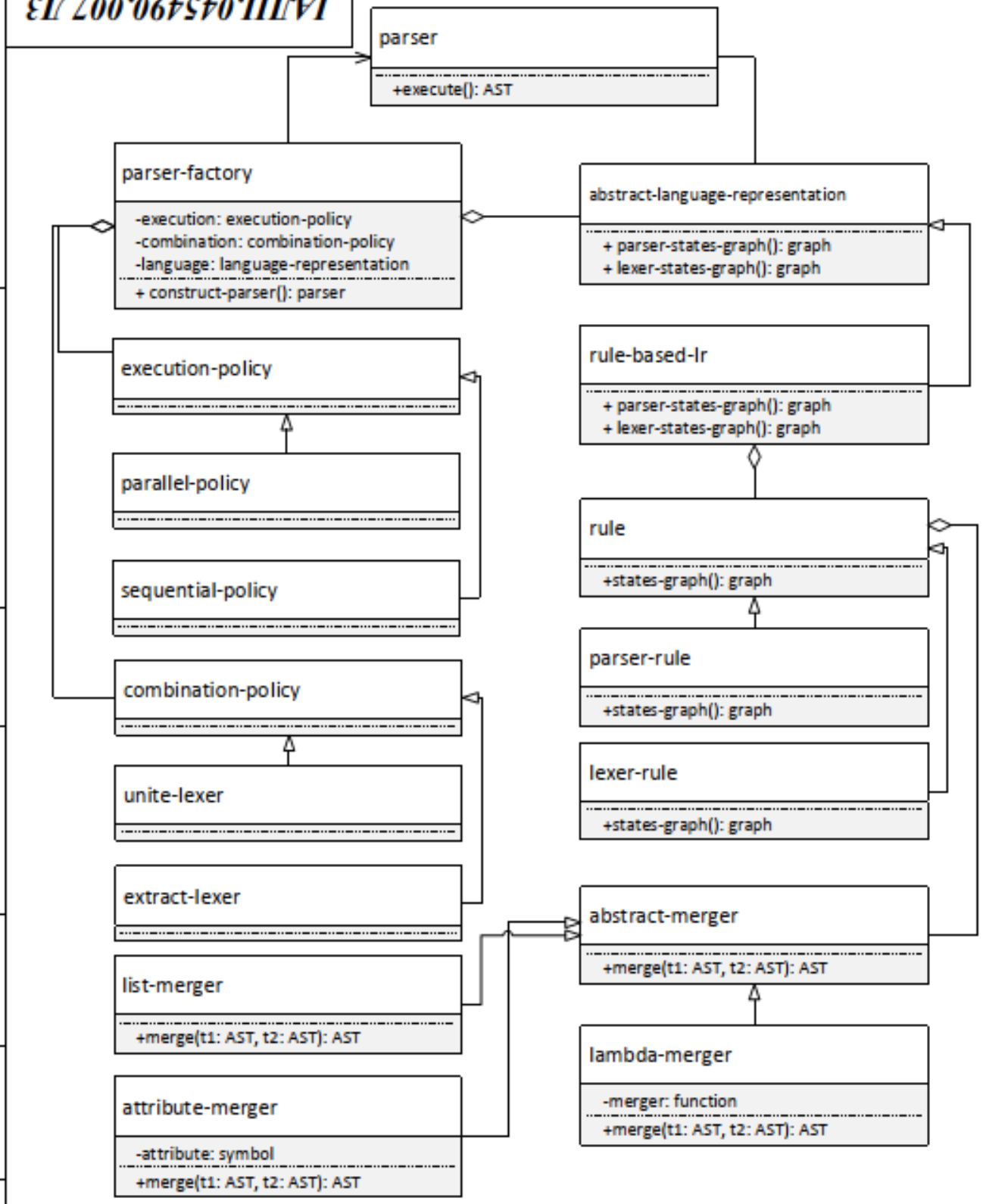
ІАЛЦ.045490.006 Д2

Структура проєкту паралельного синтаксичного аналізатора. Схема структурна

Літ.	Маса	Масштаб
Аркуш 1	Аркушів 1	

Паралельно розподілений синтаксичний аналізатор

КПІ ім. Ігоря Сікорського, ФІПМ, КВ-62



Підп. і дата | Інв. № дубл. | Взам. інв. № | Підп. і дата | № подл.

Зм	Лист	№ докум.	Підпис	Дата
Розроб.		Вовчок О. В.		
Перевірив		Зарія Ю. М.		
Т. Контр.				
Н. Контр.		Клименко Я.М.		
Затвердив		Романкевич В.О.		

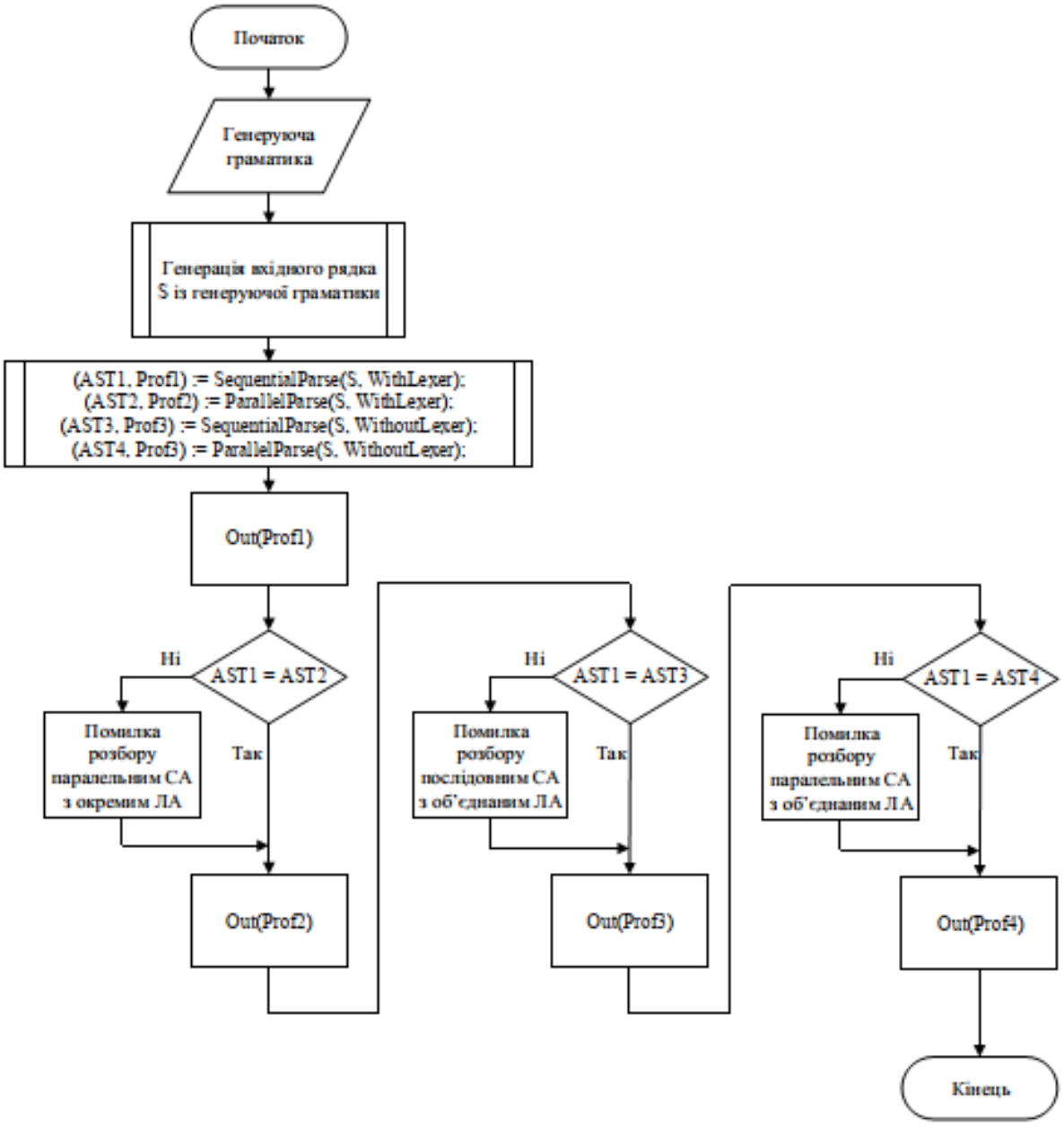
ІАЛЦ.045490.007 Д3

UML-діаграма взаємозв'язків класів. Діаграми класів

Літ.	Маса	Масштаб
Аркуш 1	Аркушів 1	

Паралельно розподілений синтаксичний аналізатор

КПІ ім. Ігоря Сікорського, ФПМ, КВ-62



Підп. і дата | Інв. № дубл. | Взам. інв. № | Підп. і дата | Інв. № подл.

Зм	Лист	№ докум.	Підпис	Дата	ІАЛЦ.045490.008 Д4		
Розроб.		Вовчок О. В.			**Алгоритм автоматизованого тестування синтаксичного аналізатора. Схема алгоритму**	Літ.	Маса / Масштаб
Перевірив		Зорін Ю. М.					
Т. Контр.						Аркуш 1	Аркушів 1
Н. Контр.		Клятченко Я.М.			Паралельно розподілений синтаксичний аналізатор	КПІ ім. Ігоря Сікорського, ФПМ, КВ-62	
Затвердив		Романкевич В.О.					